Building Production-Ready AI Agents with FastAPI, Pydantic-AI & MCP 🚀

Petros Savvakis

whoami_

** **Lead Software Engineer** @ Ethniki Asfalistiki — Tech Thirsty… shipping cloud-native FastAPI + K8s micro-services

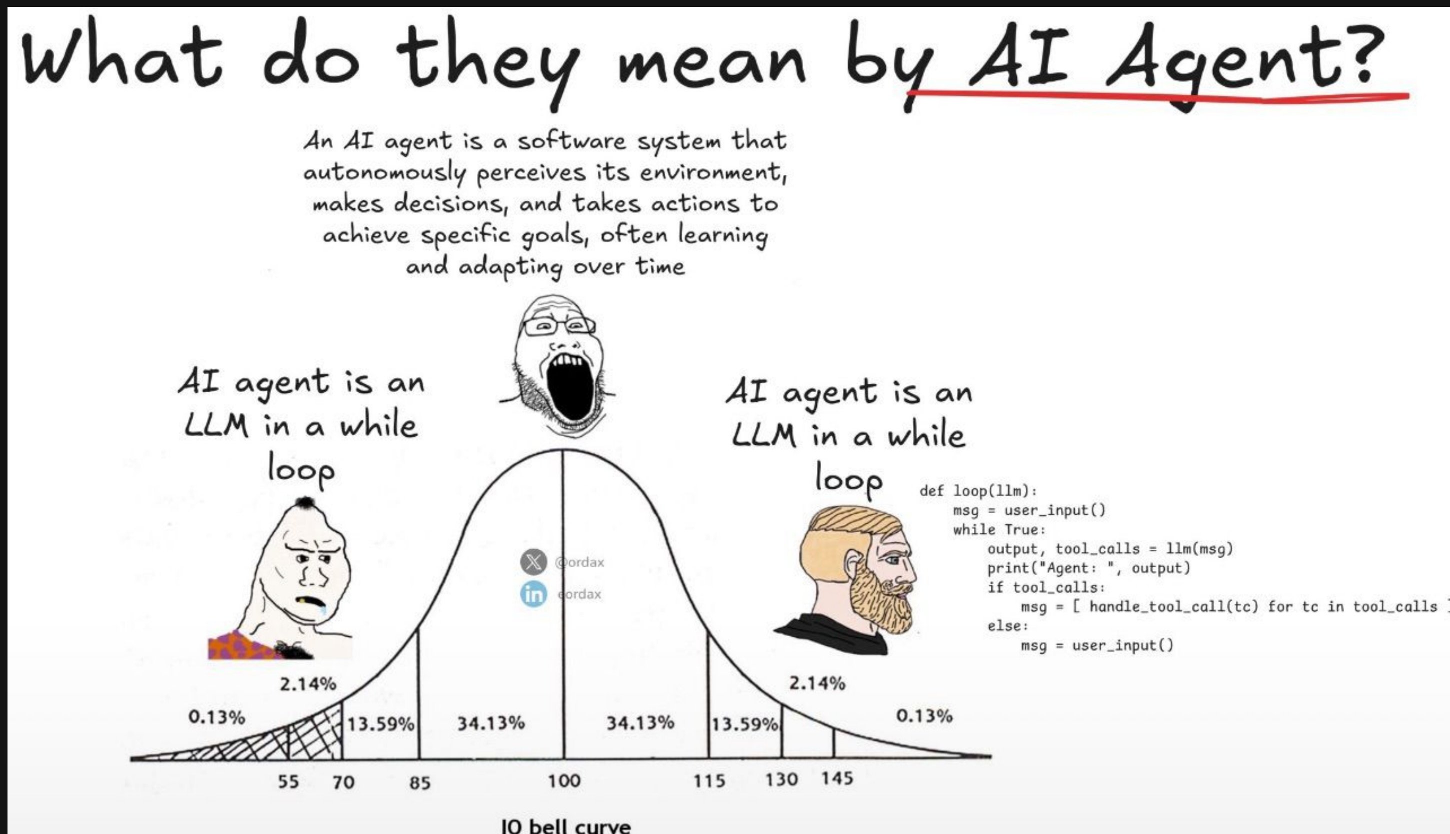
** **MSc Robotics & Electrical Engineering**; ex-PCB hacker-designer

** **OSS tinkerer & blogger** — respectablyAI, PeepDB, writing about tech at petrostechchronicles.com

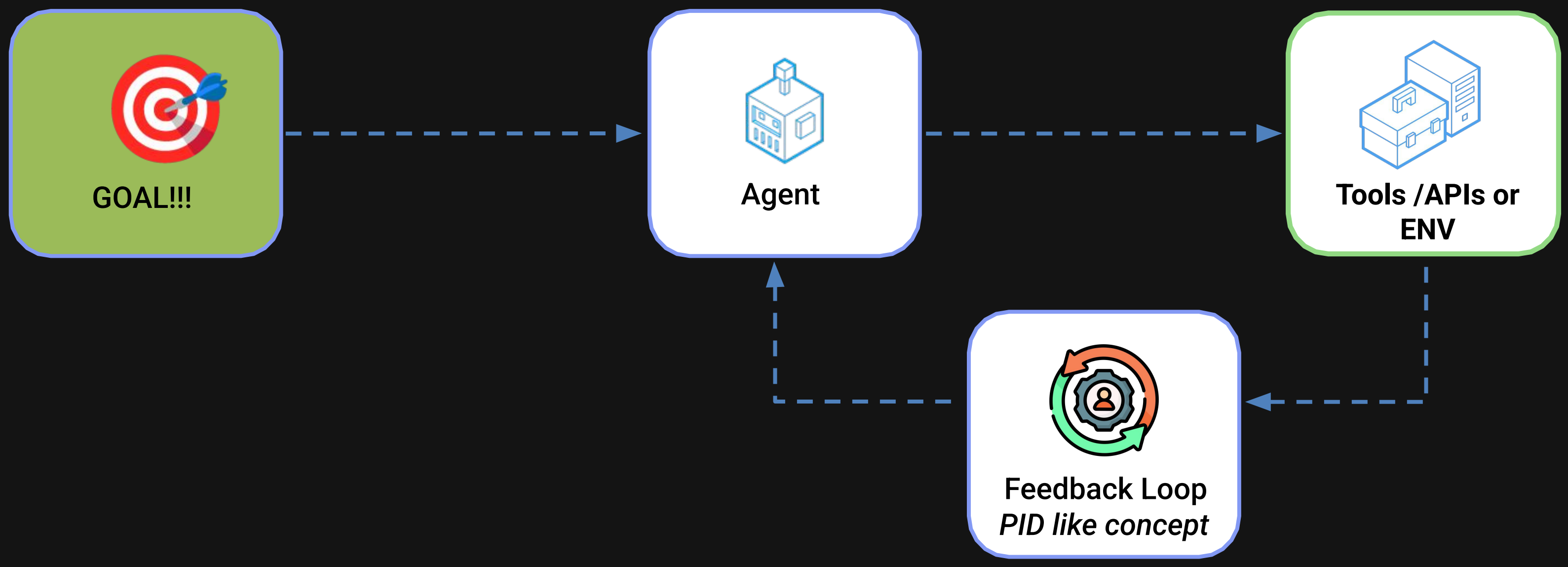
why?

WHY DO WE NEED AI AGENTS IN GENERAL?

Agents???


What do they mean by AI Agent?
An AI agent is a software system that autonomously perceives its environment, makes decisions, and takes actions to achieve specific goals, often learning and adapting over time
AI agent is an LLM in a while loop
AI agent is an LLM in a while loop
def loop(llm):
    msg = user_input()
    while True:
        output, tool_calls = llm(msg)
        print("Agent: ", output)
        if tool_calls:
            msg = [handle_tool_call(tc) for tc in tool_calls]
        else:
            msg = user_input()
0.13%
2.14%
13.59%
34.13%
34.13%
13.59%
2.14%
0.13%
55
70
85
100
115
130
145
IQ bell curve

what is AI Agent?



Agent definition:

- Execute workflows autonomously
- Make decisions based on context
- Maintain conversation history
- Can use external tools

Picture an AI assistant that retains context and executes actions—that's the essence of an agent.

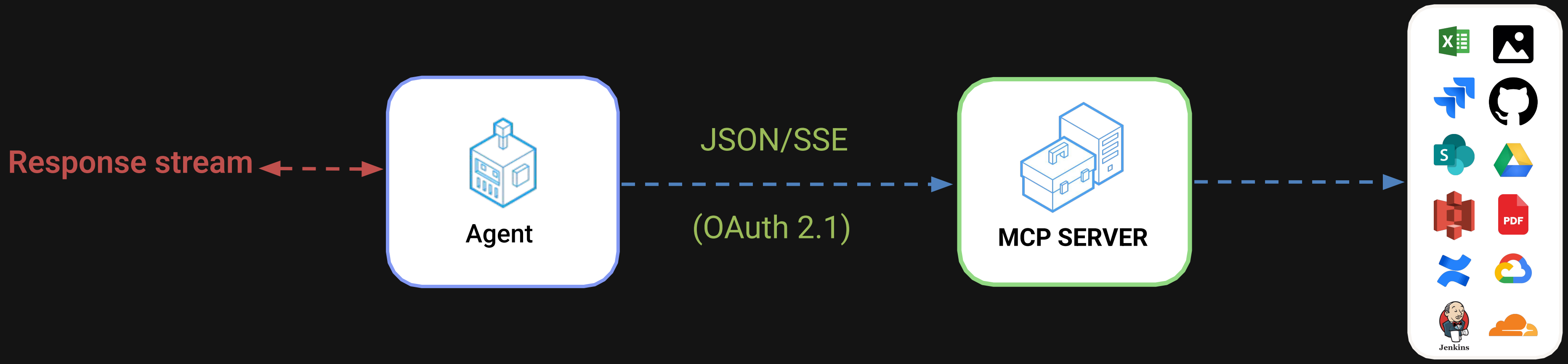
For Agent to work we need:

MCP (Model Context Protocol)

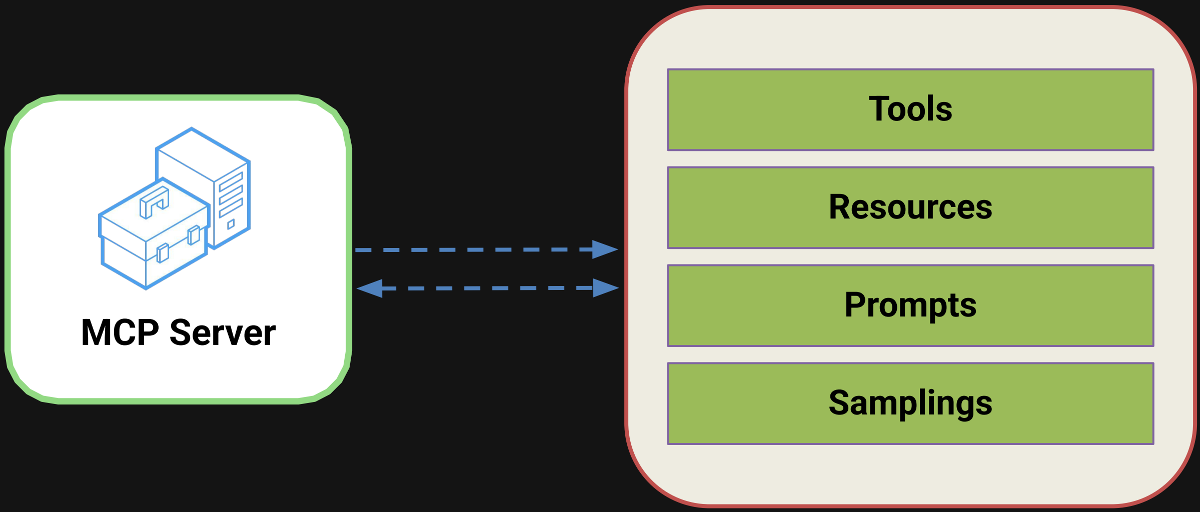
What is MCP?

- **Open protocol for AI** ↔ tool handshakes (HTTP + JSON/SSE)
- **Secure *(partially)* - using** (OAuth 2 + RBAC)
- **Language/model agnostic clients** (Python, JS, CLI)
- **Interoperability (Universal Adapter)**
  Ready-made adapters for GitLab, Confluence, Jira, Slack, DBs

why do we need MCP(in detail)?



MCP Server-tools can be...



Tools (“*get_weather*”):

```
import json, requests

# ❶ Describe & expose a callable tool on the server
GET_WEATHER = {
    "name": "get_weather",
    "description": "Return current weather for a city",
    "inputSchema": {
        "type": "object",
        "properties": {"location": {"type": "string"}},
        "required": ["location"],
    },
}
# (server would put this in the list returned by tools/list)

# ❷ Model decides to call the tool
payload = {
    "jsonrpc": "2.0",
    "id": 2,
    "method": "tools/call",
    "params": {"name": "get_weather", "arguments": {"location": "New
York"}},
resp = requests.post("http://mcp-server.example/tools/call", json=payload)
print(resp.json())
```

Resources *(project file)*:

```python
import requests, pprint, json

# List everything the server has exposed
res_list = requests.post(
    "http://mcp-server.example/resources/list",
    json={"jsonrpc": "2.0", "id": 1, "method": "resources/list"},
).json()["resources"]

# Grab the first Rust source file
rust_uri = next(r["uri"] for r in res_list if r["mimeType"] == "text/x-
rust")
contents = requests.post(
    "http://mcp-server.example/resources/read",
    json={
        "jsonrpc": "2.0",
        "id": 2,
        "method": "resources/read",
        "params": {"uri": rust_uri},
    },
).json()["content"]

pprint.pp(contents.splitlines()[:10])
```

Prompts (*re-usable code_review*):

```python
import requests, json

# Discover reusable prompt templates
prompts = requests.post(
    url="http://mcp-server.example/prompts/list",
    json={"jsonrpc": "2.0", "id": 1, "method": "prompts/list"},
).json()["prompts"]

# Fill the “code_review” template with the user’s snippet
hydrated = requests.post(
    url="http://mcp-server.example/prompts/get",
    json={
        "jsonrpc": "2.0",
        "id": 2,
        "method": "prompts/get",
        "params": {
            "name": "code_review",
            "arguments": {"code": "def hello():\n    print('world')"},
        },
    },
).json()["messages"]  # ready-to-drop chat messages
```

Samplings (*server-initiated completion*):

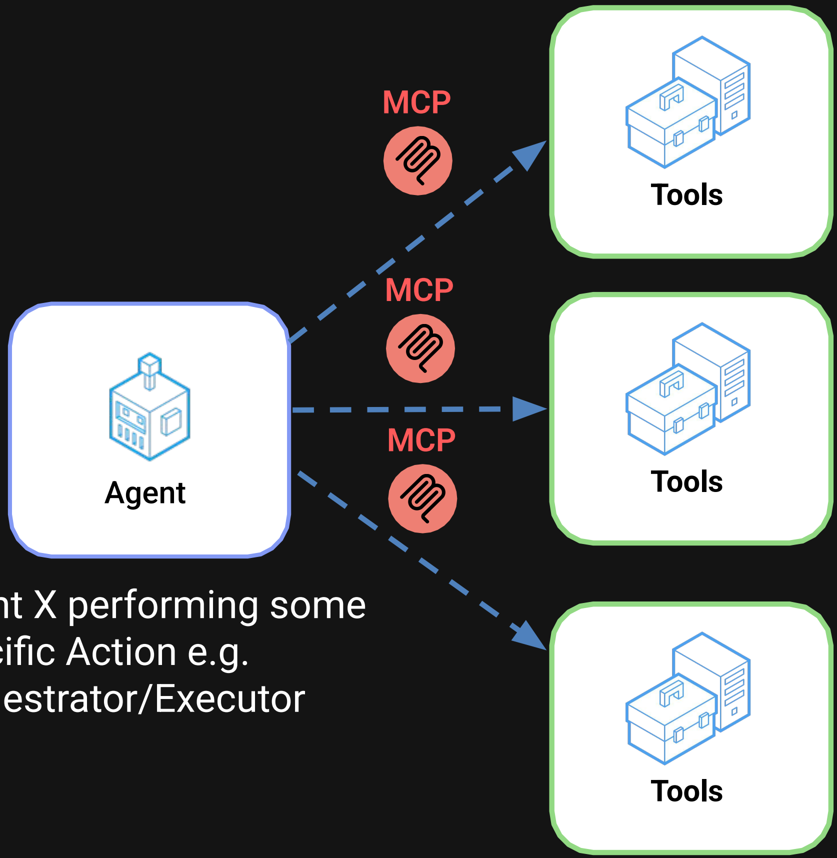
```python
import requests, json

sampling_req = {
    "jsonrpc": "2.0",
    "id": 1,
    "method": "sampling/createMessage",
    "params": {
        "messages": [
            {"role": "user", "content": {"type": "text", "text": "What’s the capital of France?"}}
        ],
        "systemPrompt": "You are a helpful assistant.",
        "modelPreferences": {
            "hints": [{"name": "claude-3-sonnet"}],
            "intelligencePriority": 0.8,
        },
    },
}

answer = requests.post("http://mcp-server.example/sampling/createMessage",
json=sampling_req).json()
print(answer["result"]["choices"][0]["message"]["content"]["text"])
```

MCP server transport options 🚦

Transport	Best-fit use case	How it works	Extra notes
stdio (local / “spawn & pipe”)	When the client can launch the server as a subprocess on the same machine *(e.g., VSCode, Cursor, local CLI tools)*.	JSON-RPC messages flow over stdin → stdout; each message is newline-delimited UTF-8.	Easiest to support; every MCP client should implement it.
Server-Sent Events (SSE)	Legacy remote transport for long-running cloud servers where you want true streaming but haven’t upgraded yet.	Two HTTP endpoints: POST for requests, long-lived GET that returns Content-Type: text/event-stream for streaming responses and server-initiated notifications.	Still widely supported, but being phased out in favor of Streamable HTTP.
Streamable HTTP	Recommended remote transport for new deployments *(Cloudflare Workers, FastAPI, etc.)*.	Single HTTP endpoint that supports: • POST for client → server messages • Optional SSE stream (same URL) for server → client messages.	Supersedes SSE; supports resumable streams and simplifies firewall / CORS setup.

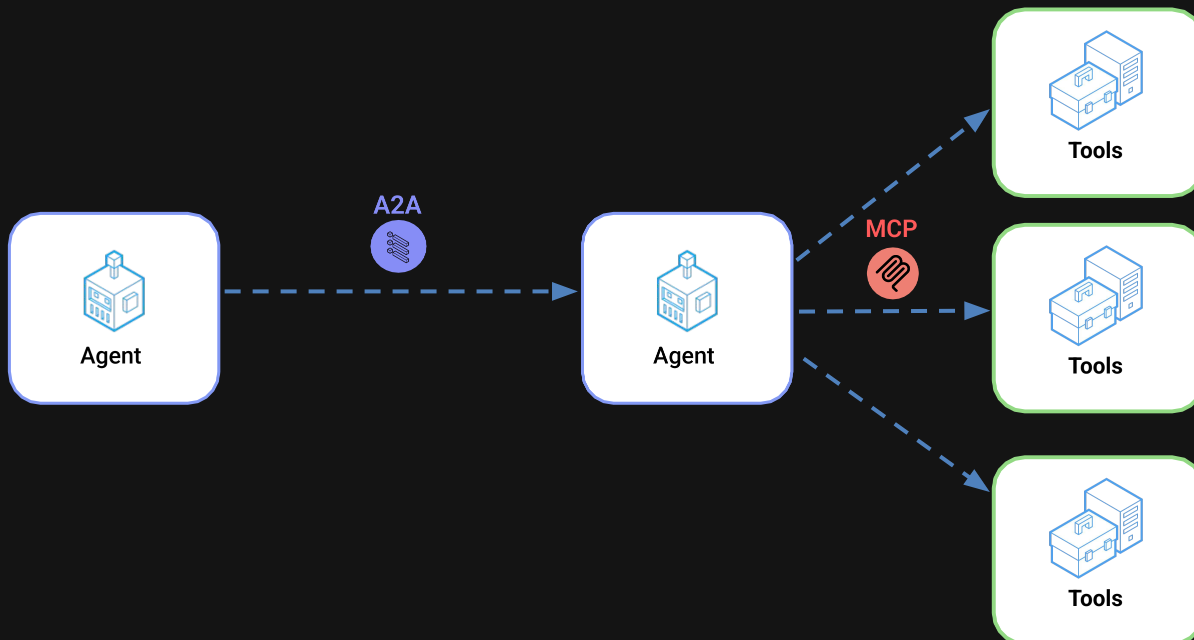
Fusing Agents with MCP

Agent with MCP Communication



Agent X performing some specific Action e.g. Orchestrator/Executor

We also need A2A… for complex pipelines



What is A2A(Agent2Agent) ?

- **Open Agent-to-Agent protocol** for direct communication, task delegation & real-time result streaming between heterogeneous AI agents (HTTP + JSON-RPC/SSE)

- Agents publish a discoverable "**Agent Card**" (ID, skills, endpoints) so peers can auto-discover and negotiate work

- **Shared security model** OAuth 2 / scoped keys with signed messages—to keep cross-vendor traffic safe and auditable

- Enables **multi-agent "swarming"** workflows that complement MCP's agent-to-tool layer (plan → execute → verify) without a central orchestrator

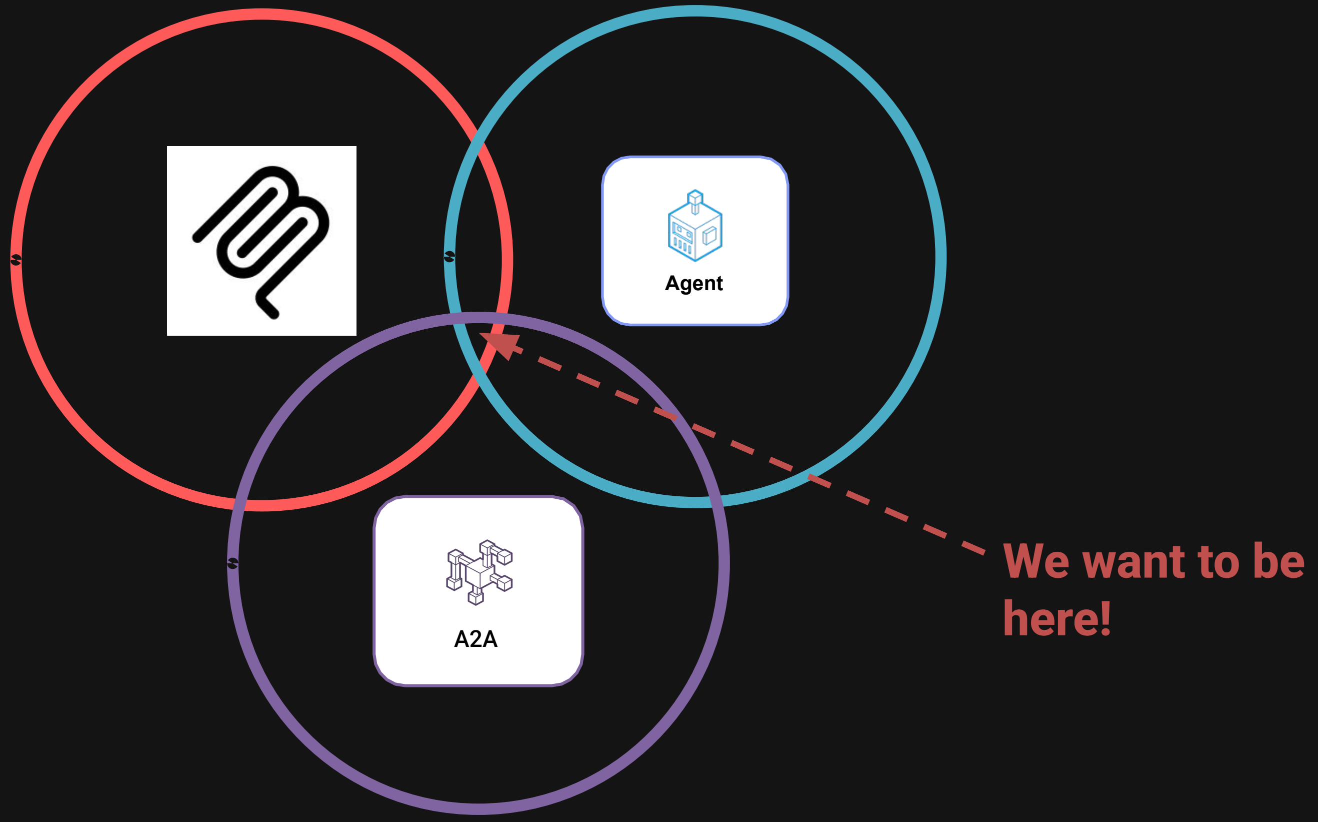
Why do we need A2A to connect multiple Agents?

- Secure Collaboration
- Task and State Management between Agents
- UX Negotiation
- Capability Discovery

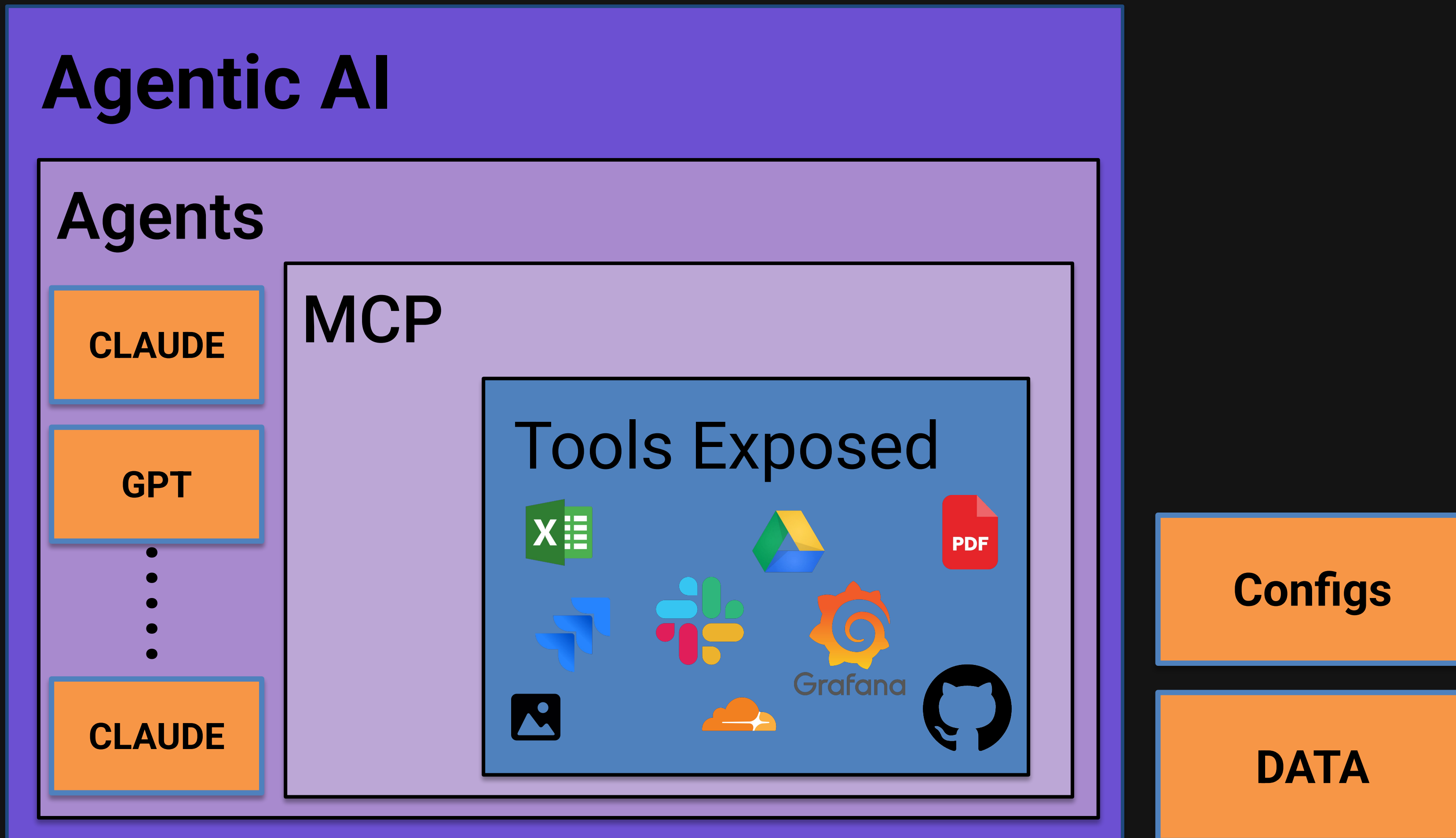
Clarifying why we need each protocol...

Feature / Protocol	MCP	A2A
Focus Area	Context Sharing	Peer Task Collaboration
Type	Context Protocol	Communication Protocol
Best Use Case	Multi-model memory sharing	Decentralized agent operations
Scalability	High with MCP servers	High in P2P networks
Complexity	High	Moderate
Standardization	Evolving	Emerging *(more early stage than MCP)*
Security Layers	Context visibility control *(poor performance security wise)*	Authenticated exchanges

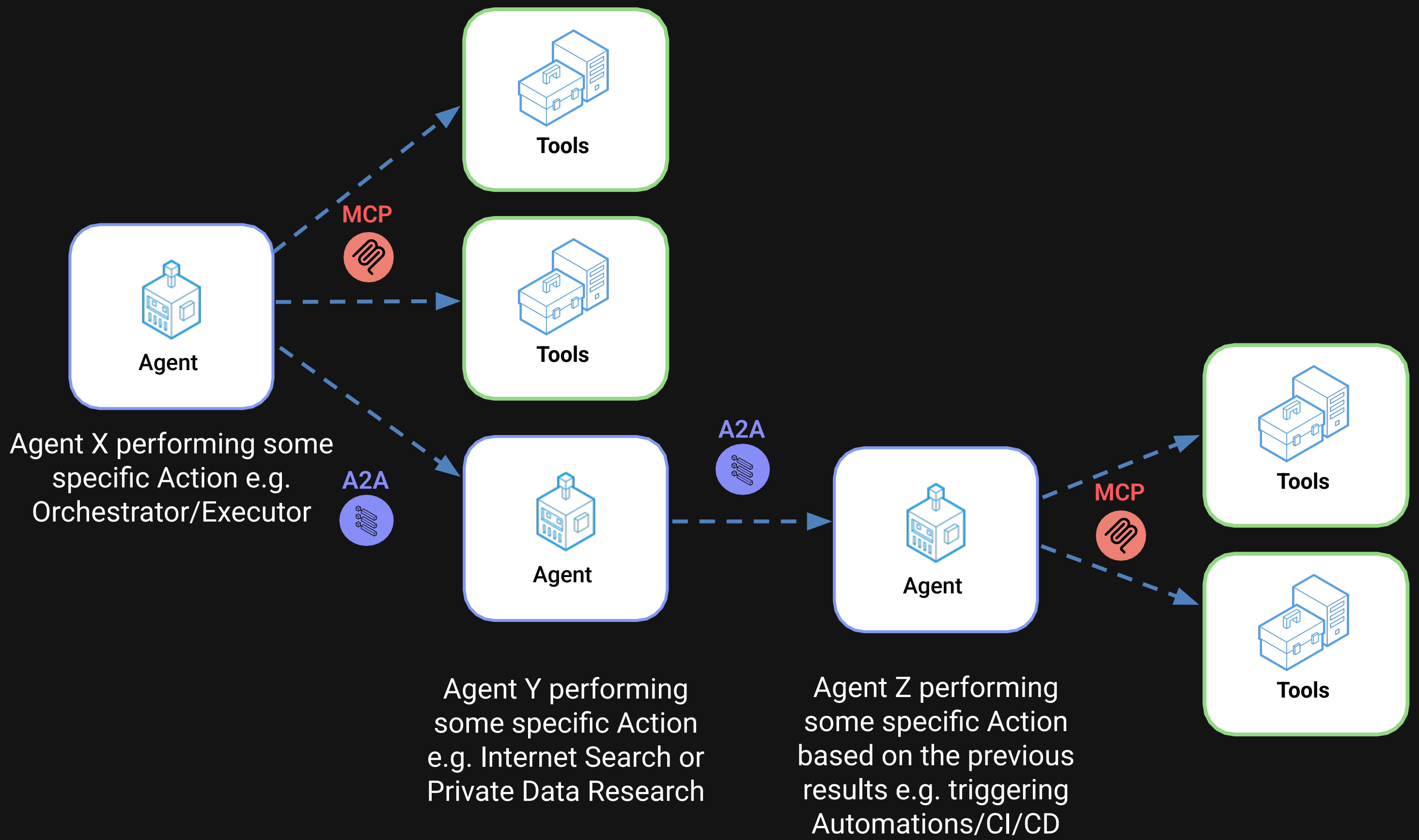
Fusing Agents with MCP + A2A



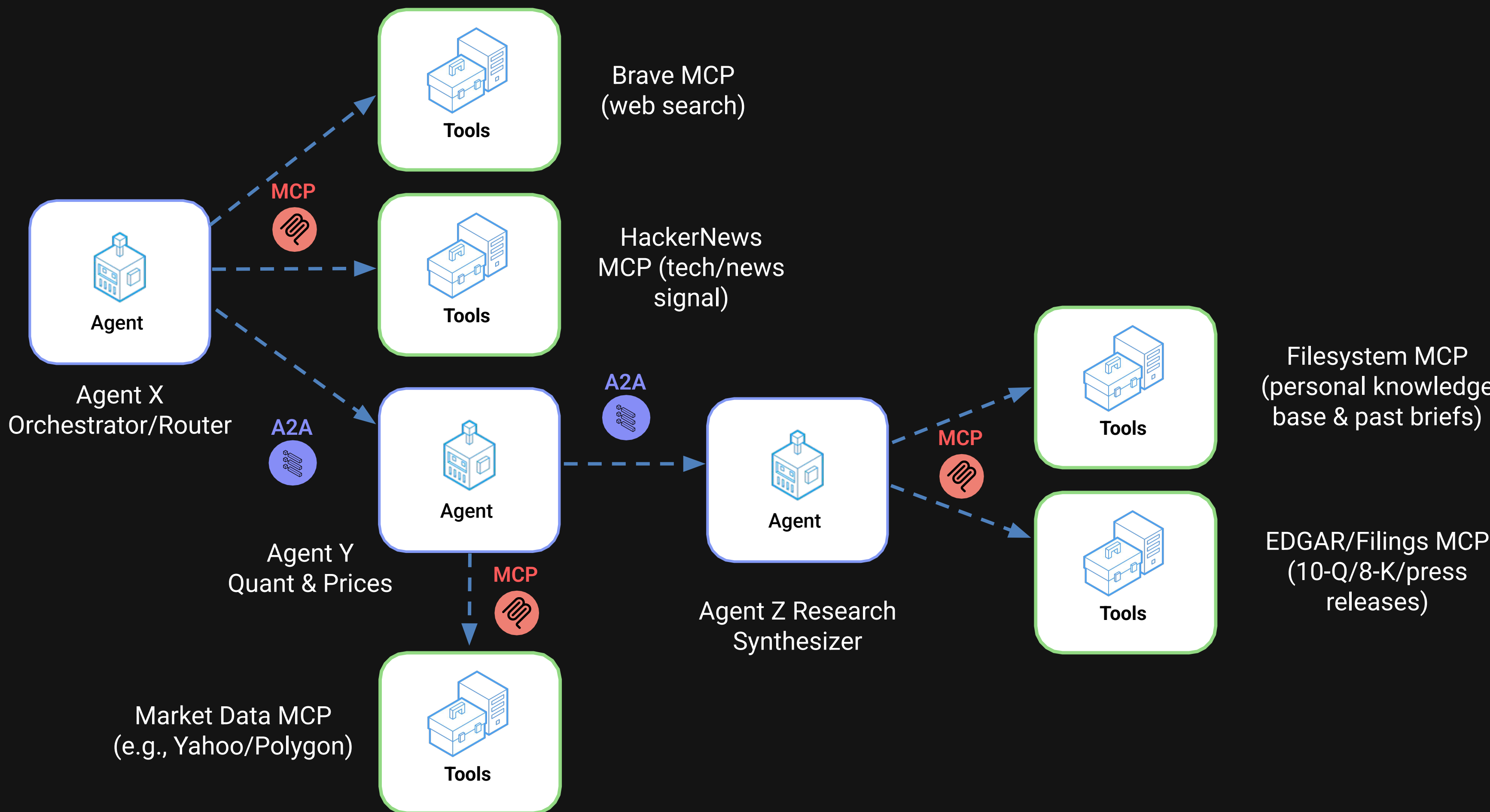
Agentic AI - Convoluted



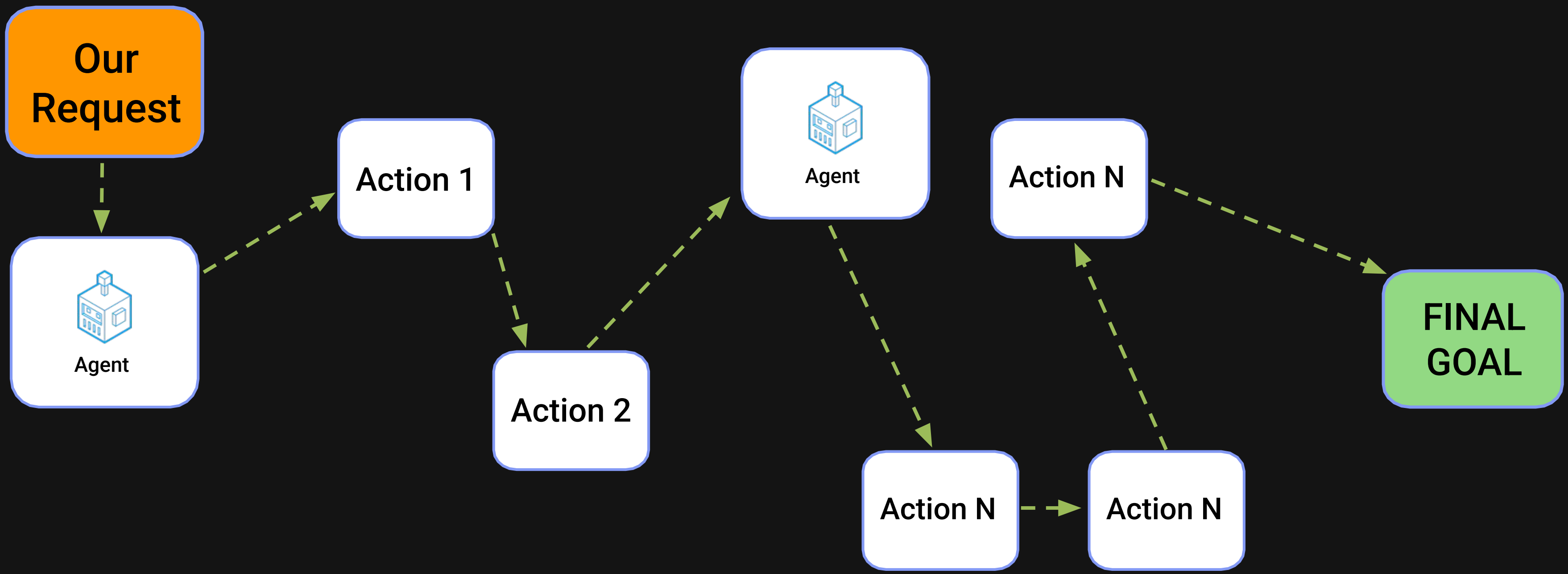
MCP + A2A multi-Agent Communication Pipeline



Pilot Use Case

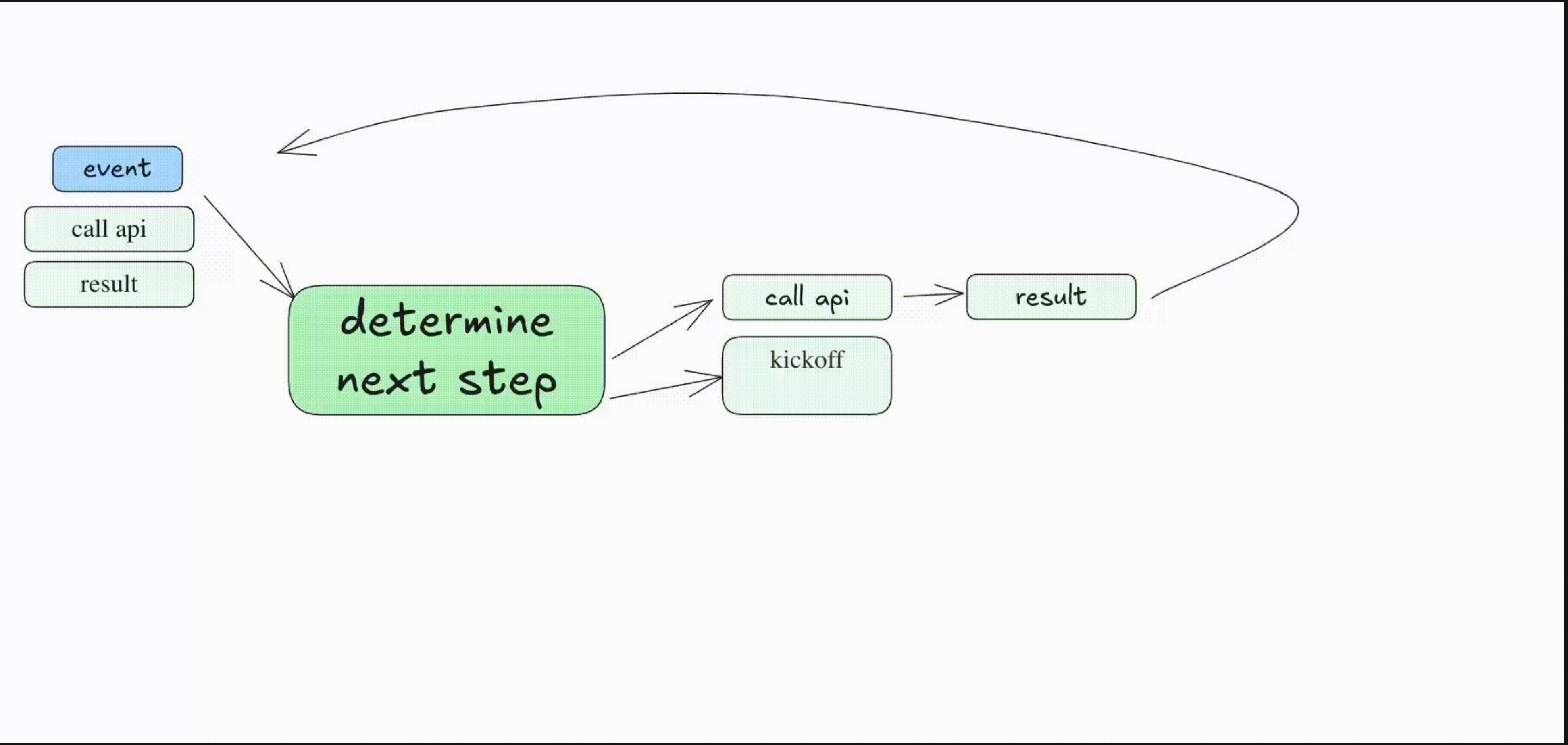


What we hope to achieve...



In reality...

In reality...



Source: *https://github.com/humanlayer/12-factor-agents/blob/main/img/027-agent-loop-animation.gif*

Now that we have an idea about all that...

Let’s build something more plausible…

Each small agent handles a focused task (e.g. summarisation or classification), making the overall system easier to debug and scale

- Structured tool calls & schemas
- Own your prompts & context
- Deterministic control flow & logging
- Human-in-the-loop triggers

FastAPI Explanation!

FastAPI is a modern, fast (high-performance) web framework for building APIs in Python.

It’s built on Starlette and Pydantic, so you get high speed and automatic validation

Key Features include:

- Very high performance (comparable to Node.js and Go or *at least trying*😅)
- Standards-based: uses OpenAPI and JSON Schema for automatic interactive docs
- Fast to code with editor autocompletion and fewer bugs

Let’s now pair it with Pydantic AI 🥳

What is Pydantic AI?

- Pydantic AI is a Python agent framework that brings the “FastAPI feeling” (type-safety, great DX, automatic validation) to Gen-AI app development
- Built and maintained by the core Pydantic team the same validation layer trusted by OpenAI, Anthropic, LangChain, etc.

Why bother using it?

- **Structured output ⇄ LLM flexibility:** Define a Pydantic model → Pydantic AI guides the LLM to emit JSON that matches it → auto-parses & validates every run (no regex hacks).
- **Model-agnostic:** Works with OpenAI, Anthropic, Gemini, DeepSeek, Ollama, Groq, Cohere, Mistral—and you can plug in any new model with a tiny adapter
- **First-class observability:** Plugs straight into Pydantic Logfire for real-time debugging and usage metrics
- **Type-safe & async-friendly:** Static type-checkers catch mistakes; supports synchronous & asynchronous runs out-of-the-box.

SURVEY TIME!!!

Let’s see what knowledge the audience has about Agents🔥

{LET'S START CODING REAL EXAMPLES...}

In practice though… it takes a whole stack to deploy Production AI systems…

When Do Agents Make Sense in Production?

📈 Value given right

📈 Probability of Success
(as it is a nonlinear system)

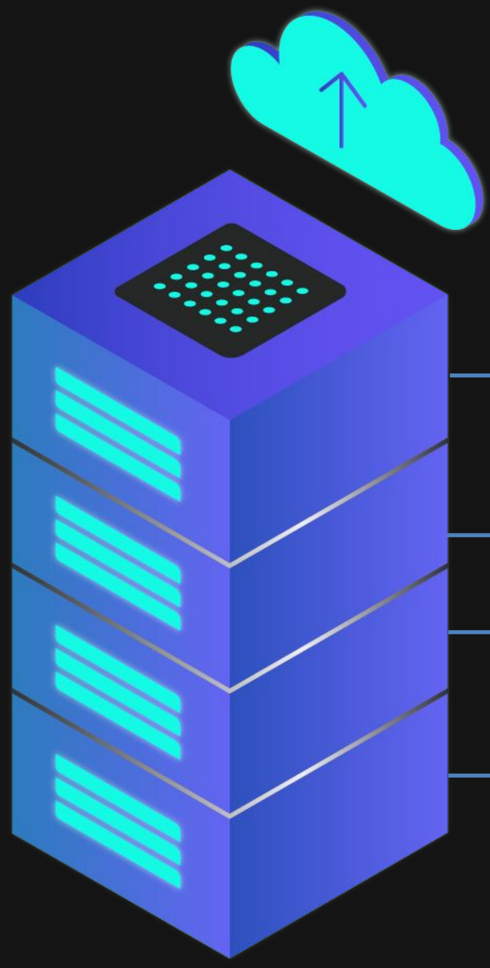
📉 Cost of getting the wrong answer

=>

$P * V - (1 - P) * C >$ 💰

Source: *3 ingredients for building reliable enterprise agents - Harrison Chase, LangChain/LangGraph*

Production-Ready AI Agent Stack



- Layer 4: Infrastructure & Security 🔒
- Layer 3: Communication Layer (A2A) 🔄
- Layer 2: Tool Integration Layer (MCP) ⚙️
- Layer 1: Agent Intelligence Core 🧠

Layer 1: Agent Intelligence Core 🧠

- **Deterministic I/O:** Pydantic schemas for inputs/outputs; strict parsing & coercion.
- **Memory:**
  - short-term (context window mgmt),
  - long-term (pgvector/Weaviate/Pinecone),
  - entity memory; TTL + purge jobs.
- **Prompt mgmt:** versioned prompts, templating, A/B variants, feature flags.
- **Tool calling:** constrained functions with JSON schema; guardrail validation before/after calls.
- **Reasoning control:** max tool-call depth, recursion caps, timeouts, circuit breakers.
- **Fallbacks:** model routing (primary/backup), offline rules for degraded mode.
- **Caching:** semantic + input hash caching (Redis) with eviction policy.
- **Evals:** automated regression evals (hallucination, grounding, toxicity, task success); golden sets.
- **Safety filters:** PII redaction, jailbreak/abuse detection, allow/deny tool lists per role.
- **Cost/latency control:** token budgeter, streaming responses, batching.
- **Observability hooks:** trace every step (LangSmith/OpenTelemetry), prompt/response snapshots, cost meters.

Layer 2: Tool Integration Layer (MCP) ⚙️

- **MCP contracts:** typed tool specs, idempotent operations, clear error codes.
- **Auth to tools:** per-tool secrets, token scoping, rotation, least privilege.
- **Rate limiting & backoff:** retry policies, hedged requests, circuit breakers per tool.
- **Data guards:** input validation, output sanitization, schema checks; content provenance tags.
- **Timeouts:** per-tool SLAs; cancel + cleanup on over-time.
- **Streaming & chunking:** large payload handling (multipart, resumable, pagination).
- **Sandboxing:** FS/network isolation for Filesystem/Code tools; allowlisted paths/hosts.
- **Auditability:** tool call logs (who/what/when/why), request/response hashes.
- **Versioning:** pin tool server versions; backward-compatible changes; canary new tools.
- **Local vs remote:** health checks, readiness probes; failover to alternate MCP endpoint.

Layer 3: Communication Layer (A2A)

- **Protocol:** explicit agent contracts (roles, capabilities, message schema, step limits).
- **Routing:** planner/router agent with deterministic policy + heuristics; loop detection.
- **State shared-nothing:** pass minimal, signed state; avoid hidden globals.
- **Delivery guarantees:** at-least-once via queue (NATS/Kafka/RabbitMQ) with dedup keys.
- **Idempotency keys:** for replays/retries across agents.
- **Traceability:** correlated request IDs across agents; OpenTelemetry spans.
- **Access control:** per-agent RBAC/ABAC; capability tokens for allowed tools.
- **Escalation paths:** human-in-the-loop handoff; stop/go approvals for risky actions.
- **Cost/latency budgets:** per-conversation ceilings; kill-switch when exceeded.
- **Testing:** multi-agent simulations, chaos tests (drop/slow/duplicate messages).

Layer 4: Infrastructure & Security 🔒

- **Runtime:** Docker/Compose/K8s with resource limits, HPA, node affinity (CPU/GPU).
- **Networking:** mTLS between services, service mesh (Linkerd/Istio) for retries/CB.
- **Secrets:** Vault/Secrets Manager, rotation, per-env scopes, no secrets in images.
- **Storage:** Postgres for state, object store for artifacts, vector DB with backups & PITR.
- **CI/CD:** supply-chain security (SBOM, image signing, vuln scans), canary + blue/green.
- **Monitoring:** metrics (p95 latency, token/s, cost/s, tool error rate), logs, alerts with SLOs.
- **Data governance:** PII cataloging, retention policies, delete/trace requests, encryption at rest.
- **Compliance:** audit logs, DPIA where needed, data residency, DLP on egress.
- **Resilience:** multi-AZ, backup/restore drills, dependency SLOs, graceful degradation.
- **Cost mgmt:** per-tenant metering, anomaly detection, budgets & alerts.

Core Production Building Blocks

UI Interface	**Chat UI** (Slack/Teams)	**Web/Mobile Interface**	**API Gateway** (Kong, Tyk, NGINX/Envoy, Traefik, Cloudflare Gateway)		
Orchestration	**FastAPI**	**LangChain - LangServe**	**Pydantic-AI** (MCP client adapters)	**Custom Orchestrator** (Celery/RQ/Temporal,HTTP/ WebSocket routers; gRPC(Protobuf))	
Prompt Management	**Prompt Construction** (Pydantic-AI tools, Guidance, Instructor, LMQL, DSPy)	**Versioning** (Git, DVC for prompt artifacts, LangSmith runs)	**Chat History** (Postgres (JSONB), SQLite, MongoDB, RedisJSON)	**Context Management** (LlamaIndex, Haystack, LangChain RAG)	
Memory & Tools	**Short/Long-term Memory** (mem0, LangGraph state stores)	**Vector DBs** (pgvector (Postgres), Qdrant, Weaviate,, Pinecone, Chroma)	**Redis** (Redis Stack, Redis Streams for events, redis-rate-limit)	**Prompt Cache** (PTCache, Redis/SQLite LRU, LiteLLM cache middleware)	
Communication Layer (A2A)	**Protocol Contracts** (MCP, JSON Schema, gRPC/Protobuf, Avro (Kafka))	**Routing & Delivery** (NATS, Kafka, RabbitMQ, Celery/Redis)	**RBAC/ABAC** (Keycloak, Auth0, Ory (Kratos/Keto), JWT/OIDC)	**Observability** (OpenTelemetry, Logfire SDK,Prometheus, Grafana)	
Tool Integration Layer (MCP)	**Typed Contracts** (MCP SDKs, Pydantic models)	**Sandboxing** (E2B sandboxes, Firecracker, Docker seccomp/AppArmor)	**Audit Logs** (ELK (Filebeat/Logstash/ES/Kibana), Grafana Loki, CloudTrail)		
Open Source Models **(Registries - Hugging Face)**	OpenAI	Anthropic	Llama	DeepSeek	Mistral
Infrastructure & Security	Docker/K8s	mTLS Networking	Secrets Manager	Monitoring/Alerts	CI/CD Security

Case 1: Personal Usage (Solo / Homelab) 👨‍💻

Resources & Models

- 🖥️ Hardware: 1× consumer GPU (RTX 3090/4090, 16–24 GB VRAM) with strong CPU
- 🧠 Models: LLaMA-2 7B, Mistral 7B, Gemma/CodeGemma (4–30B)
- 💾 Storage: Local SSDs, model weights (4–20 GB each)
- 🗄️ Data: SQLite/Postgres for RAG over personal docs

Stack (Minimal)

- FastAPI + Pydantic-AI + MCP servers (Filesystem, Web Search)
- Docker/Proxmox for self-hosting

Use Cases

- Personal coding/chat assistant/news aggregator
- RAG over PDFs, notes, configs
- Experimental agents

Case 2: Small Team (5–10 Developers)👨‍💻👨‍💻

Resources & Models

- ⚖️ Infra: Shared GPU workstation (24–48 GB VRAM) or hybrid (local + spot cloud GPU)
- 🧠 Models: LLaMA-2 13B, Mixtral 8×7B (quantized for VRAM fit)
- 💾 Storage: Central DB (Postgres/Mongo), model weights (20–40 GB)
- 🗄️ Vector DB for docs (e.g. Chroma, Weaviate)

Stack (Minimal)

- FastAPI services in Docker / Compose
- vLLM or Hugging Face TGI for serving
- Observability: Prometheus + Grafana / ELK
- RBAC + secrets mgmt (Vault, env vars)

Use Cases

- Internal bots (docs search, workflow assistants)
- Team-wide RAG over shared knowledge
  CI/CD integrated agents

Case 3: Large Teams (>10, Enterprise-Grade)👨‍💻👨‍💻👨‍💻

Resources & Models

- ☁️ Infra: Cloud GPU clusters (A100/H100, CoreWeave, GCP, Azure)
- 🧠 Models: LLaMA-2 70B, Mixtral 8×7B, domain-fine-tuned models
- 💾 Storage: Distributed (S3, Snowflake, BigQuery), multi-TB vector DB
- 🌐 Networking: NVLink/InfiniBand for multi-GPU inference
- The list goes on….

Stack

- Multi-agent orchestration (MCP, LangGraph, Azure Agent Service)
- K8s + CI/CD pipelines (staging → prod)
- Observability: Datadog, OpenTelemetry, Arize AI
- Security: Enterprise RBAC/ABAC, compliance (SOC2, GDPR)
- The list goes on….

Use Cases

- Customer support copilots
- Financial/data QA agents
- Multi-modal assistants across business units

Source: *Those resources are examples from real deployment use cases such as (Brave, Perplexity, Intuit, IBM, etc.).*

Key Takeaways! 🚀

Agent = context + actions + tools

MCP = plug-and-play interoperability

FastAPI = production-grade APIs & orchestration

PydanticAI = type-safe scaffolding for agents

Stack ≠ LLMs -> Infra + Monitoring + Security...

Let’s see the survey Insights 📊

Thank you!!!

Feel free to connect with me at **Ln**:

or feel free to visit my blog:

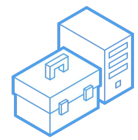
Or even better come and chat with me!!! 😃

Icons

MCP

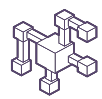
A2A

Agent



Tools

Feedback Loop
PID like concept



A2A

PDF

Jenkins